4

RADC-TR-89-359
Final Technical Report
February 1990

REAL TIME AUTOMATIC PROGRAMMING

The MITRE Corporation

Gary A. Cleveland, Richard L. Piazza, Richard H. Brown

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC-TR-89-359 has been reviewed and is approved for publication.

APPROVED: [illegible]

DOUGLAS A. WHITE
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: [illegible]

IGOR G. PLONISCH
Directorate of Plans & Programs

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-359
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS (City, State, and ZIP Code) Burlington Road Bedford MA 01330	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-87-C-0001
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62702F / PROJECT NO. MOIE / TASK NO 73 / WORK UNIT ACCESSION NO 10

11. TITLE (Include Security Classification)
REAL TIME AUTOMATIC PROGRAMMING

12. PERSONAL AUTHOR(S)
Gary A. Cleveland, Richard L. Piazza, Richard H. Brown

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Oct 86 TO Sep 87	February 1990	44

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence, Real Time
Knowledge-based Systems, Software
Automatic Programming

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes the MITRE developed automatic programming system ISFI and the attempt to use this system in the production and modification of small to moderate sized real-time programs. ISFI is a knowledge-based automatic programming system that has been developed at MITRE during the past 5 years. The ISFI system is based upon the theory that networks of constraints are an appropriate knowledge representation tool for many domains, and illustrates a number of techniques such as propagation and transformation that facilitate inference in constraint networks as a method for code synthesis. Keywords:

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Douglas A. White	22b. TELEPHONE (Include Area Code) (315) 330-3564 22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

1 Introduction

1.1 History

ISFI is a knowledged-based automatic programming system that has been developed at MITRE during the past 5 years. The ISFI system is deeply rooted in Brown's thesis [1]. This thesis claims that networks of constraints are an appropriate knowledge representation tool for many domains and illustrates a number of techniques such as propagation and transformation that facilitate inference in constraint networks as a method for code synthesis. One of the primary differences from other work on constraint networks [2] is that in writing programs using ISFI, one can be said to "compile" networks of constraints whereas other constraint satisfaction schemes "interpret" the networks.

The implementation for the thesis was limited to synthesizing numerical programs in MacLisp. The ISFI project seeks to extend the previous work in two important ways. First, objects other than numbers are considered. Real world objects have structure that may change during the execution of the synthesized program; one must provide for dealing with these changes. Secondly, the ISFI system incorporates a general view of programs and programming languages. As such, the system deals with a more general model of programming languages instead of exploiting the strengths of a single language.

1.2 Overview

In the view of the designers and implementors of ISFI, an automatic programming system is one that accepts as input a non-procedural specification of behavior and produces as output a program that exhibits the specified behavior. Simply put, the eventual user of such a system should be able to state what he wants to happen (without telling how it should happen - that's programming) and be given a program that makes it happen. Our primary goal is to improve software maintenance, including changes in requirements, debugging of existing features, and the addition of new features. The emphasis is on the incremental changes to a software system rather than on the first time production. Producing any large, complicated system for the first time will remain a demanding task [3,4]. Our approach emphasizes the use of various knowledge bases and the ability to account for how and why those knowledge sources were used. We feel such information is essential to the process of software maintenance.

In the automatic programming domain, the world is described by specifying the existence of objects with specific properties, attributes, and relations to other objects. Behavior of any system takes two forms: 1) the input/output behavior and 2) the non-monotonic behavior. In the first form, some input objects are used to compute some output objects (the output objects are created according to the relationships given by the program specification; see [5] for philosophical discussion). In the second form, some set of relationships among objects is changed. For our model, this change can always be described by adding (deleting) an object to (from) some property or attribute of a given object. These two forms of behavior are not mutually exclusive.

Functionally, ISFI accepts a set of objects and relationships as the specification for the program. Some of the relationships are treated as true and are used in the computation of objects that are generated by the synthesized program. Other relationships are declared in such a way that the synthesized program causes them to become true; these relationships specify side effects. ISFI's output is a program in the target language that exhibits the desired behavior both in performing side effects and in computing objects for output.

Given this model of the world, we see that the input/output behavior of a program can be specified simply by indicating that certain objects are either inputs or outputs of a program. The relationships specified among all of the involved objects can be used to compute the output objects. For non-monotonic behavior, however, where relationships among objects must be changed by the synthesized program, we need some mechanism for stating that at a given time a relationship is true and that at some later time it becomes false. Thus, we need to state a temporal ordering for the veracity of relationships among objects. To fill this need, a finite state diagram is used, where objects exist in states and directed transitions between states provide the time ordering.

A point to emphasize is that ISFI is not tied to any given target language. The representations of objects, their behavior, and the description of the synthesized program are all free of language dependent considerations. Only a very small section of ISFI's code generator actually deals with the target language's syntax, naming conventions, scoping, parameter passing, and other considerations. Currently, ISFI can generate code in Common-Lisp and C.

2 Terminology

2.1 Objects

First of all, ISFI needs to deal with classes of objects as well as individual objects. A hierarchy is useful in organizing the object classes. For simplicity, we limit ourselves to a strict hierarchy (A Kind Of tree or AKO tree) [6] rather than pursue a lattice structure (tangled hierarchy) [7,8] that allows an object class to inherit from multiple immediate ancestors. A strict hierarchy allows us to state facts like, "An integer is a kind of number," but prevents statements such as, "An elephant is a kind of four legged animal and a kind of gray object."

A mechanism called *roles* is used to describe attributes, parts, properties, and qualities of objects. It is similar to mechanisms in other frame-like systems. A role of an object (class) is "filled" by one or more objects of some designated object class, indicating that the given attribute has a value related to the object filling the role. Thus, to resolve our multiple hierarchies problem we could say that, An elephant is a kind of four legged animal[1] and that, Elephants have a color role that is filled by some object whose value indicates gray. This method of declaring one principal hierarchy and describing other attributes by a separate mechanism can be somewhat arbitrary but is usually adequate.

Each class of objects has associated with it some number of roles that declare potentially relevant attributes, parts, properties, and qualities for each object in the class. These roles will also pertain to each object of each sub-class of the original class. In this way, roles are inherited and we can discuss, for example, the engine sizes of trucks as well as those of cars, buses, and other types of motor vehicles.

We also can specialize a role of an object class when it inherits the role from a class higher in the AKO tree. For example, a DATSUN-280Z object class may have a role called ENGINE-TYPE (filled by an object from the possible ENGINE-TYPES). A sub-class may be called DATSUN-280ZX-TURBO and expect an object of class TURBO-ENGINES (a sub-class of ENGINE-TYPES) in its ENGINE-TYPE role. For the sub-class, more information is introduced for the role although we would like to consider it to be the same role as that used by the super-class.

Roles manifest themselves in ISFI as *role descriptors* which are associated with two object classes: the *assoc-class* and the *member-class*. The assoc-class represents the object class that has the part, attribute, property, or quality described by the role descriptor. The member-class represents the object class that fills the role.

An important distinction between ISFI's role descriptors and the attributes, slots, parts, etc., that are found in most other knowledge-based systems is that role descriptors combine in a single mechanism the notions of parts, attributes, slots, etc., whereas many other systems tend to treat these separately (see [9,10,11,6]). Another important distinction is that role descriptors may contain either an object or a collection of objects. Conceptually, a role that holds a single object is a special case where the collection is of size one. Collections allow the intelligent cooperation between data structures that hold multiple objects of the same type and real world objects that have multiple-valued attributes. For example, a dog kennel usually has more than one dog. In ISFI we can express that the role descriptor DOGS is a collection of objects of class DOG. Further, we can now index and order this collection when it is implemented on a data structure that supports indexing and ordering.

2.2 Relationships

Now that we know approximately what objects are and how they are described, we will consider the ways in which we wish to deal with relationships among objects. Two requirements must be met for the relationships in the system we are discussing. First of all, the relationships mechanism should be general.

[1] In this paper, any text that appears in sans-serif font is an example of an element of an ISFI specification or of the knowledge bases of ISFI. Any text that appears in typewriter font is an example of code that ISFI generated or something that it uses during code generation, i.e., code templates.

This requirement stems from the fact that programs are written for nearly every possible domain. The second property that relationships must have is that they must be transformable into some operations upon the involved objects.

As objects are of certain classes, constraints (representing relationships) are of certain *constraint types*. Constraint types are not, however, arranged in any sort of hierarchy. Constraint types contain information about the type of objects that the constraint relates and a number of *constraint laws* that specify how some objects can be computed from other objects involved in the relationship. One can think of constraint laws as computational interpretations of constraints. Constraint laws may also specify how to perform certain side-effects.

As mentioned in the previous section, roles are used to state an implicit relationship between two objects (two objects are related by the fact that one fills a role of the other). Sometimes it is necessary to make this whole/part relationship explicit, so there is also an IN-ROLE constraint type.

A special relationship that deserves attention is that of equality. Specifically, there are five kinds of equality in ISFI:

1. Two objects are equal if they are the same identical object. This relationship is implicit.
2. Two objects are equal if they are connected to each other by the EQUALITY constraint.
3. Two objects may be viewed as the same object but be mapped to different data structures in the synthesized program. In these cases, there may or may not be a set of constraints that defines a conversion between the two representations.
4. Two objects are not equal in either of the first two ways but have enough roles in common to be considered equal. For example, the many incarnations of Morris the Cat and Lassie were always treated as being equal to the other incarnations. Deciding when "enough" roles are the same is another area of consideration all to itself. This topic will not be covered in this paper.
5. Two objects are equal but exist in different contexts.

Only two representations of equality are available in the ISFI system – *wires* and the equality constraint. Because the equality constraint is just a constraint like any other, special equality considerations (for example: 3, 4 and 5 above) must be handled through wires. Wires are discussed below.

2.3 Non-Procedural Specification

We choose to represent objects and their relationships using *networks of constraints* [2,1,12,13]. Relationships, therefore, are *constraints* to be enforced upon objects or *nodes*. One could envision the networks as a first-order logical statement, but we do not take a "theorem proving" approach to automatic programming.

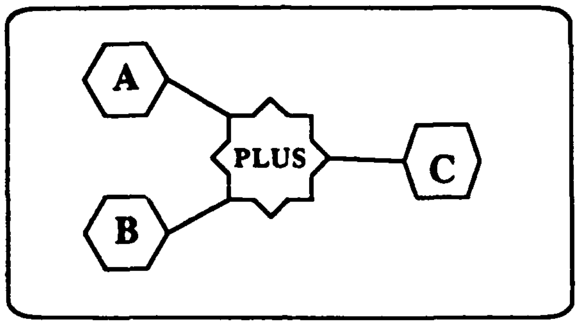


Figure 1: An Example Network

A simple example of a network is found in figure 1.[2] The constraint type for addition connects to three numeric objects (designated the augend, addend, and sum *connections*) and represents that the sum of two of the numbers (augend and addend) equals the third (sum). Obviously, given the values of the augend and addend nodes, the value of the sum node can be computed. This corresponds to one of the laws for the addition constraint type. Less obviously, there are two more laws that describe computations for the values of the augend and addend using the subtraction operator. Thus, the constraint describes the relationship among the three nodes, not just a single computation. Because networks are *non-directional*, it is necessary to indicate the input nodes and output nodes in order to make the specification complete.

Additionally, two nodes can be connected using wires. The semantics of a wire is that when one node has a value, the other node has the same value. This formalism is used to express the special equality considerations mentioned previously. Wires can be directional (*interstate* wires) or non-directional (*intrastate* wires). Interstate wires are discussed in the next section.

A network embodies a *non-procedural specification* of some behavior. Saying that the sum of two numbers is a third number has an obvious computational meaning. On the other hand, saying that a taxi cab ought to be yellow in no way determines how to go about making it yellow. Some series of mechanisms must work to use this information in a reasonable way when trying to form an appropriate computation. While it is true that every relationship to be used in synthesizing a program must be transformable into a computation, the initial set of relationships does not necessarily form a computation. The original specification is in a non-procedural form. Without the ability to deal with non-procedural specifications, any system that claims to be an automatic programming system is, at best, an ultra-high level compiler.

[2]In the figures throughout this paper, nodes will appear as hexagons and constraints as eight-pointed stars.

2.4 States

A ***state transition hierarchy*** organizes each network of constraints. The state transition hierarchy adds structure to the specification of nodes and constraints by partitioning the nodes of the network of constraints into states. Nodes are said to "live" in a state. The state transition hierarchy merges two common formalisms: finite state machines and hierarchies. Viewed as a hierarchy, states provide a rough scoping mechanism. Viewed as a state (context) mechanism, states provide for the separation of mutually inconsistent statements (non-monotonicity). Viewed as a transition diagram this formalism provides a minimal specification of a program's gross flow of control.

One branch of the state hierarchy is *active* at any given time, which means that any relationship between objects that are in the branch is *true* (i.e., the constraints among nodes are to be enforced). Constraints themselves do not live in states. The values of objects can be passed from branch to branch on interstate wires, which is the one of the special types of equality ISFI provides. From a practical standpoint, interstate wires are connected to the objects that must be saved before switching contexts.

The state transition hierarchy has three important properties:

- *Provides necessary vocabulary.* The state transition diagram provides a way of specifying such requirements as "Event A happens before event B."
- *Provides necessary functionality.* The ISFI system can derive scoping, control flow, and other information from the state specification.
- *Intuitive.* State diagrams are reasonably easy to create and understand when specifying programming problems.

The state transition hierarchy addresses three basic specification issues: side effect management, time ordering, and environments, which are discussed below.

2.4.1 Side Effects Management

The state transition hierarchy provides ISFI with a simple but effective context mechanism. This mechanism is necessary because constraints express what is true at a given time (or in a given context). The state transition hierarchy allows the specifier to introduce different contexts to separate mutually inconsistent facts. For example, to specify that a traffic-light's color changes from red to green, we would specify in one state that the color attribute had value red and in a later state that the color attribute had value green (see figure 2). These two statements obviously conflict if asserted in the same context. This topic is discussed in more detail below.

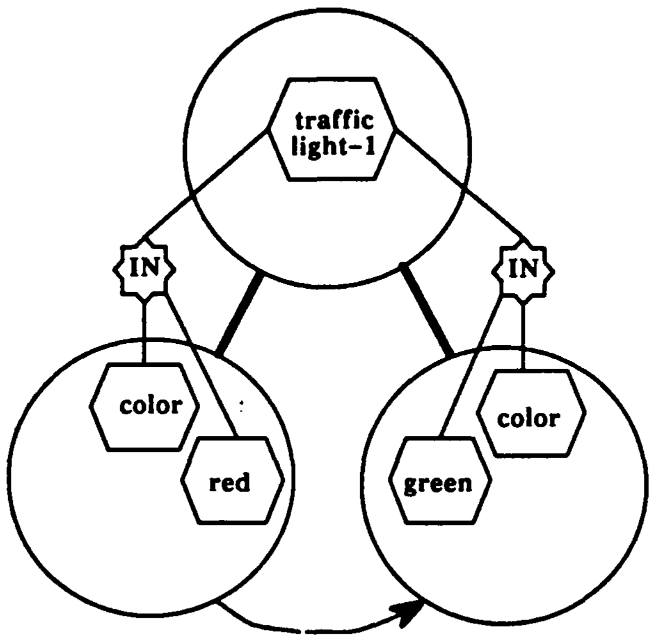


Figure 2: Specifying a Side Effect

2.4.2 Time Ordering and Control Constructs

The transitions of the state hierarchy provide the ability to specify a time order of changes. This is a necessary component of any specification and is inherent in addressing change (the world before and after).

In addition, the concepts of conditionals, recursions, and common subproblems seem basic to a specification regardless of the abstraction level, and can be indicated using transitions. Time ordering of this type is independent of change, but is nevertheless essential to a specification.

For example, to specify a decision point (branch) in the control flow, we would use three states (A, B, and C) and two transitions (from A to B and from A to C). The two transitions would be enabled by the two possible values of a boolean node in the state A. Higher numbers of branches can be obtained by using different object classes for the decision node (e.g., traffic-light-colors = {red, green, yellow} would give a three-way branch, see figure 3).

2.4.3 Environments

One basic rule that ISFI enforces for all specifications is that the computations of objects in a given state can not depend upon objects in lower states in the hierarchy. This corresponds to the basic rules of lexical scoping - one can only access up the hierarchy. In ISFI, we have adopted the model of block-structured programs and lexical scoping for our representation of programs and their environments (see figure 4).

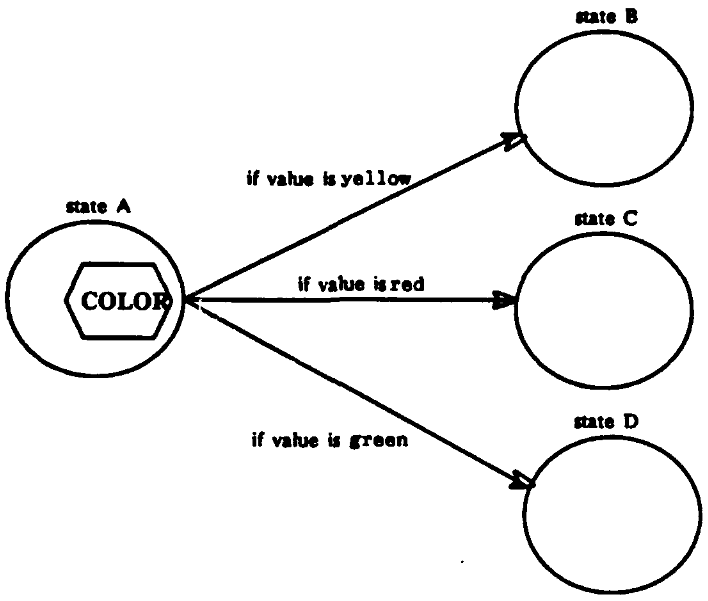


Figure 3: Specifying a Decision Point

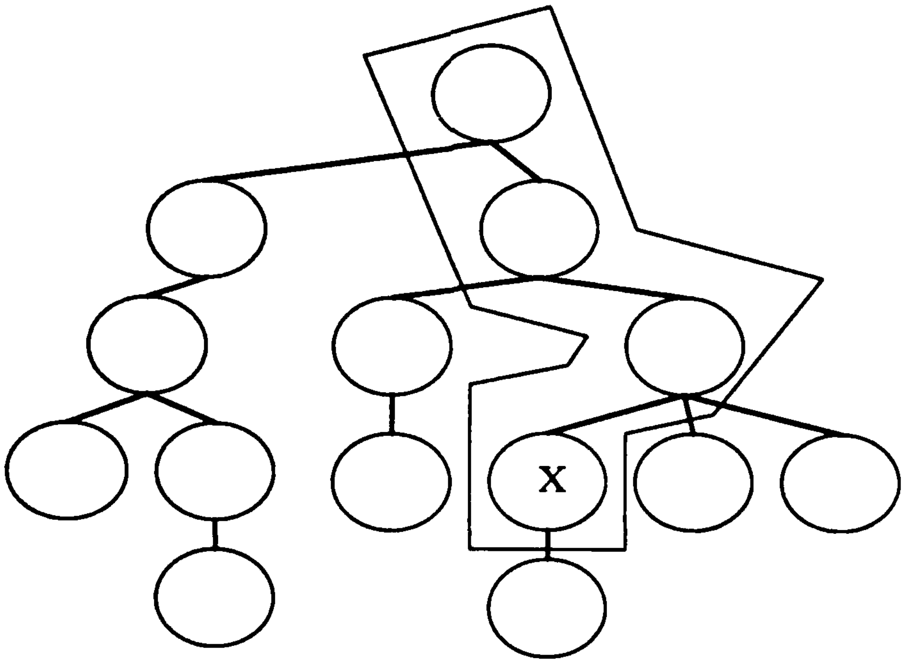


Figure 4: All Objects in Outlined States are Available in State X

2.5 The Knowledge Bases

We encode knowledge in two knowledge bases: the domain knowledge base and the programming knowledge base. Although the ISFI system does not make any strict categorization along these lines, the distinction is useful.

2.5.1 The Domain Knowledge Base

The domain knowledge base is usually expressed in a group of networks that hold generic knowledge about objects in the domain. Note the use of the principle that knowledge need not be declared many times in order to be used many times. Each fact of domain knowledge is declared once and can be used on demand in many places. The kinds of knowledge found here are:

For-all Knowledge: This is a fact that is true for all objects of a particular object class. Such relationships take the form, "For all objects of a certain class, such and such relationships hold." For example, we might want to declare that All taxi cabs are yellow. This declaration is not a default that can be overridden by other information lower in the AKO tree. This information is true for all taxi cabs forever (unless we change our mind and the original declaration). Thus, if we make a statement such as, All birds can fly, we had better mean it.

This is the only type of For-all knowledge that ISFI can currently accept. Noticeably lacking is the ability to make statements such as, "For all numbers greater than one ...," and "For all possible traffic-light colors ...," and "For all members of this set ..." This would be a natural and desirable extension to the ISFI system.

Consequence of Side Effects: This is a special subset of the For-all knowledge about classes and is used when a side effect is performed on an object of a given object class. These facts can be stated as, "When a role of an object is changed, these relationships may need to be changed." For example, the relationship between fahrenheit and celsius temperatures can be used to maintain the celsius temperature whenever the fahrenheit temperature is changed.

Transformations: This is a set of IF-THEN rules where both the IF and the THEN parts are expressed in the constraint network formalism. Having found the IF pattern in a specification network, the THEN pattern can be added. Transformations are primarily useful in adding new computation paths to a network (see Figure 6).

Constraint Types, Object Classes and Roles: ISFI treats all constraint types, object classes and their roles equally; none are "built in". This fact insures that adding new constraint types or domain objects and their attributes is a straightforward process.

2.5.2 The Programming Knowledge Base

The programming knowledge includes:

Basic Operations: Basic operations are the language-independent descriptions of operations that ISFI supports. This is easily expandable and includes, for example, addition of numbers and negation of booleans.

Code Generators: Code generators encode the code string templates that implement the basic operations in each supported programming language. This also includes the string templates for implementing the operations supported by building blocks and variables. For example, Lisp numbers support addition with the string template "(PLUS <first-number> <second-number>)" and Lisp variables support assignment statements with the string template "(SETQ <name> <value>)".

Representations and Implementations: Representations map from abstract object-classes to specific language-dependent implementations (data structures). In essence, representations state how to implement a class of objects and many representations can be derived automatically.

Most of the programming knowledge is not specified in networks of constraints but is encoded in the ISFI system either declaratively or procedurally depending upon its use. For example, the string templates are encoded declaratively while the decisions used in introducing local variables are encoded procedurally.

Only the primitive data types such as floating point numbers are hand-coded. The behavior of complex data structures is specified in networks of constraints and used by ISFI to program other examples that use the data structures. ISFI works in this way because most data structures can be modified in various ways depending on how they are used. For example, a hash table has a number of options regarding the hashing function and collision handling. As implemented in ISFI, data structures can be modified by changing the specification of their behavior in the networks of constraints.

3 Mechanisms that Manipulate Relationships

Assuming a rich set of object classes and constraint types it is possible to create networks of constraints that not only represent the domain knowledge but also specify the desired behavior of various programs to be synthesized. It is up to the automatic programming system to use the specification networks as the starting point for creating a program that exhibits the desired behavior.

If some computation path can be found from the input nodes to all of the output nodes, then it is possible to generate code from the specification network. *Constraint propagation* is the mechanism used to find such a path. However, specification networks may be incomplete in the sense that such a computation path may not exist. The domain knowledge bases must be used in these cases. In fact, most of the mechanisms described in this section are used if the primary mechanism, constraint propagation, fails. In addition, these mechanisms are used for other purposes such as deriving more efficient code. All of the mechanisms are supervised by a control program [14] that decides which mechanism should be used and how.

Most of ISFI's mechanisms deal with the networks of constraints. To enable a successful propagation, the control program can heuristically identify the problem areas in the network and try adding knowledge about some of the nodes. These mechanisms perform operations such as copying groups of nodes and constraints from one network to another or simplifying the configuration in a given network.

3.1 Propagation

Propagation, ISFI's primary mechanism for dealing with networks of constraints, attempts to produce computations for objects in some nodes given computations for objects in other nodes. Propagation always starts at some set of nodes (such as inputs and constants) that have computations (also called *values*) and attempts to find computations for other nodes until the process can go no further. New computations are derived from existing computations through the application of constraint laws. Note that computations for side effects are generated in exactly the same way as computations for outputs.

Values in the nodes contain the computation information generated during propagation. Values maintain dependency links to other values that reflect the chain of computations that leads to a given value. The information stored in values is used in the eventual code generation.

Every constraint is an instance of some constraint type. Associated with the constraint type are a number of laws that can be used by the constraint to compute some objects from others. That is, constraint laws can be used to determine the value of a node based on the values of other nodes. Law application is the primitive mechanism that propagation uses. When a constraint is considered, ISFI tries to apply each of its laws in turn. Each law expresses how to construct values for certain *output* nodes, given certain *input* nodes. In addition, it indicates which nodes are involved in a side effect. In general, there can be many laws for each constraint type, given the various combinations of input, output, and side effect connections. A particular law may not use some connections.

The application of a constraint law traces the following course. The first order of business is to discover if all nodes involved with the law have values (output nodes may already have values, otherwise, an *empty* value is created for them). The search for values is constrained in a number of ways. Most notably, the accessing and computation of values in nodes follows a number of rules concerning the state hierarchy. For example, the value of a (input) node can only be accessed when ISFI is "in" that node's state or below it in the state hierarchy. The value of a (output) node can only be computed from the node's state or above. A node's role descriptor can only be mutated from a state that is strictly below the node's state.

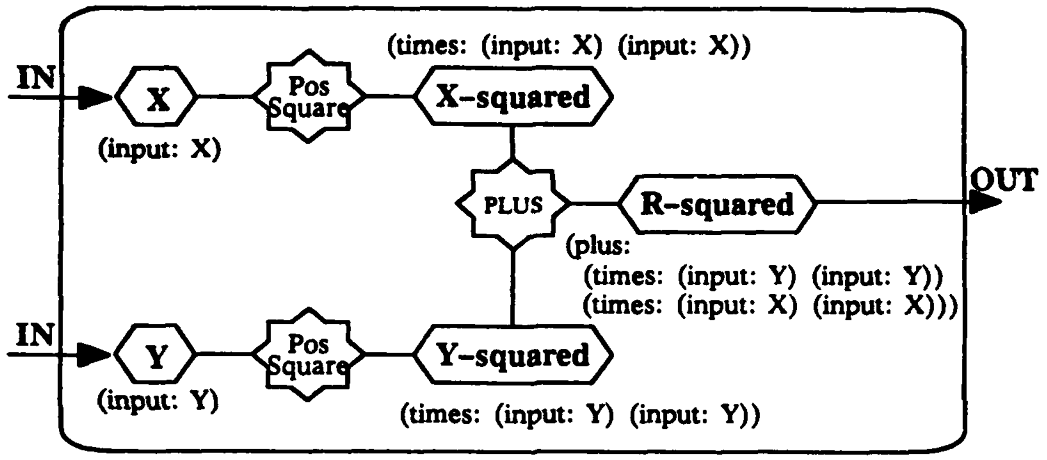

IN
X
(input: X)
Pos Square
(times: (input: X) (input: X))
X-squared
PLUS
R-squared
OUT
(plus:
(times: (input: Y) (input: Y))
(times: (input: X) (input: X)))
IN
Y
(input: Y)
Pos Square
Y-squared
(times: (input: Y) (input: Y))

Figure 5: A Network After Constraint Propagation

After all these conditions are met for a constraint law, the *algorithm fragment* of the law is used to determine what form the computations take in the output and mutation values. An algorithm fragment declares what operations should be performed on the inputs to achieve the desired results and does so without reference to how the values are implemented! For example, the addition operator may be applied to numbers regardless of their precision; the sort operator may be applied to any linear sequence of elements that supports sorting (array, linked list, binary tree, etc.). The input values for an operation are updated to show that they were used in the computation.

At any time in the application of the law, a value may not be available or some condition may not be met. Such a failure causes the law application to abort. This failure indicates that the current propagation chain terminates. Most law applications do fail, usually due to unavailable input values.

Although, the goal of propagation is to find a value for all the output nodes, the process does not necessarily terminate as soon as this has occurred. The most inexpensive computation is desired, where expense is associated with the value, and based on the operations that it encompasses. There might be other paths in the network from the input nodes to the output nodes that represent a cheaper computation. Propagation must explore all paths in the network to assure that it has found the cheapest path. For this reason, propagation terminates when the queue of constraint laws to apply is empty. ISFI's propagation avoids combinatorial explosion by forward chaining only on successful chains. When two chains meet, an evaluation of merit is performed that decides which chain should continue and in this way the unnecessary chains die off quickly. Among other things, this prevents cycles from occurring in the dependency graph that is built from the computation chains. All in all, enough computation paths die out so that propagation does not grow unwieldy in large networks.

Figure 5 describes a network after constraint propagation has taken place. The input nodes are X and Y and the output node is R-squared. Parenthesized expressions represent values.

3.2 Inheritance

As mentioned previously, the domain knowledge base contains For-all facts about the various objects. Inheritance is the mechanism that makes use of this type of knowledge. Given an object for which we need more information, we can inherit the information from the object's class and from all the classes above (superior to) that class. Inheritance is used to copy portions of network from the domain knowledge base into the network currently being solved. This use of inheritance is discussed in [9] and contrasts with systems that use inheritance as defaults.

Inheritance needs to be a controlled activity. First, it is undesirable to have redundant information in the specification. For example, if an object of class NON-NEGATIVE-INTEGER were already connected to a constraint that declared it to be greater than or equal to zero, we would not want the information to be repeated. Secondly, it may be undesirable to inherit all of the knowledge from all of the super-classes before having some indication that the knowledge is needed. This problem is alleviated because ISFI can inherit knowledge selectively, progressing up the class hierarchy only when necessary.

Additionally, as more and more nodes and constraints are copied into the network, they often form some (fairly common) configurations that we call *doublets*. Given such configurations, ISFI can deduce that certain nodes must contain the same value, and therefore can be merged. In this way, the network can actually become smaller. Doublets are discussed in detail in Brown's thesis [1].

3.3 Transformations

In ISFI (and in other program synthesis systems [15,16]), a rule application mechanism transforms sets of relationships with no computational interpretation into sets with such interpretations. The rules will not look quite like those used in classical AI systems although they will still have two parts corresponding to the hypothesis and consequence of classical rules. In this case, both parts of the rules are networks of constraints with the meaning that if the hypothesis is present then the consequence can be inferred. An example of a transformation is, Given two numbers A and B, if A+A=B then 2*A=B. In other words, we can express such common and powerful rules as, Double negatives drop out (see Figure 6), and Multiplication of integers can be expressed as addition.

Our system differs from most rule-based or transformation systems in that we never discard a relationship that still holds among the involved objects. For this reason, our form of transformation always *add* relationships rather than replace them. There is no need to throw out the knowledge that A added to itself equals B just because we also know that A multiplied by the number 2 is also equal to B.

Transformations, like inheritance, are used to copy domain knowledge in the form of networks from one place to another. The hypothesis is a pattern network that is matched against the network in which the rule is being applied. If the match is successful then the conclusion network is copied into

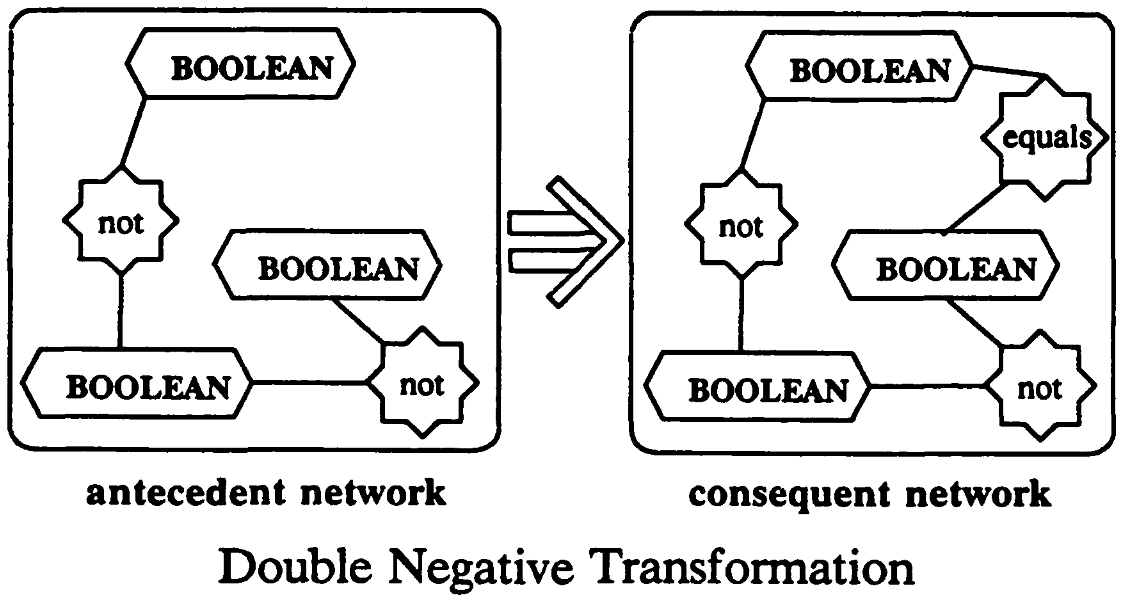


Figure 6: An Example Transformation

the original network. This copying is done in exactly the same way that the copying is done for the inheritance mechanism.

As with inheritance, the problems of applying transformations in a network center on finding nodes that already exist and on avoiding doublets. Additionally, there may be states in both the pattern and target networks making the matching process more complex. Finally, some transformation patterns can contain "wild cards" that match network structure with certain attributes.

Transformations are primarily used to resolve propagation failures. This is accomplished by copying the entire consequent network into the current network when the pattern specified by the antecedent network is matched. In this way, new relationships are deduced from existing relationships in the network. This has the effect that new computation paths are introduced into the network. In the case that a computation path already existed, transformations can be used to introduce cheaper paths leading to a better synthesized program.

Unlike the previous two mechanisms, transformation application can suffer from combinatorial explosion. Since each transformation adds some fragment of network to the existing network, the number of transformations that match grows with every successful application. To avoid forward chaining through endless networks and transformation rules, ISFI needs to maintain some knowledge about what a given transformation is likely to accomplish and then use a strategy component (discussed below) to decide which transformations are most likely to achieve a given set of goals. For example, if propagation fails because an output node can not be computed, then applying the "commutativity of addition" transformation should not be attempted because such an application can not cause any new nodes to become computable.

3.4 Side Effects and Consequences

Side effects in ISFI are represented as *mutations*. Our early formulation of mutations is discussed in [17]. A mutation is defined by the node being mutated, the role descriptor, the node that represents the contents of the role, and the operation type (insert or delete). In order to handle mutations successfully, we assume system omniscience; the only changes for which the ISFI system is responsible are those that the ISFI system causes. Most mutations appear in the synthesized code as data structure modifications and I/O operations.

The hard problems involving side effects have always concerned managing the consequences and timing of those side effects. Managing the consequences is classically known as the *frame problem* [18]. Until now the best solution to this problem lay in *reason maintenance* [19,20] but those solutions are not easily applied because the ISFI system has the more strenuous task of predicting what relationships will need to be maintained and writing a program that performs the maintenance when the actual data becomes available. The ISFI system benefits by accessing the domain knowledge to limit the scope of the potential change.

Our system detects and manages the consequences of a given side effect by considering the relationships that are changed by the side effect. By our definition of a side effect, the only kind of relationship that can change is that represented using roles. This limitation (of representation - not of expressive power) is what makes the frame problem tractable. In ISFI, the frame problem may be stated as, "Given some changes to the properties (roles) of some objects, what changes to the properties of other objects need to be made to maintain consistency?" (see figure 7) Consistency is specified by some set of constraints.

An example of tracking consequences can be found in the domain of military map display programs. Given some object-classes ENEMY-AIRBASE, BOOLEAN, and DISPLAY-COLOR and some roles such that ENEMY-AIRBASES have DISPLAY-COLORS and a TARGETED? property (a BOOLEAN), we can state that the DISPLAY-COLOR depends upon whether the ENEMY-AIRBASE is targeted or not. With this much knowledge available, suppose we ask the ISFI system to write a program to change the TARGETED? property of an ENEMY-AIRBASE. When the program is written, ISFI must also find an appropriate computation that changes the DISPLAY-COLOR to keep the set of relationships consistent. The general scheme is that from a set of changing role values and from the relationships expressed in the knowledge base, the automatic programming system generates a set of roles that may be affected and forms an appropriate computation for each.

In ISFI, the problem of managing consequences boils down to collecting the needed constraints from the domain knowledge base and copying them into the problem network. Normally, all constraints are

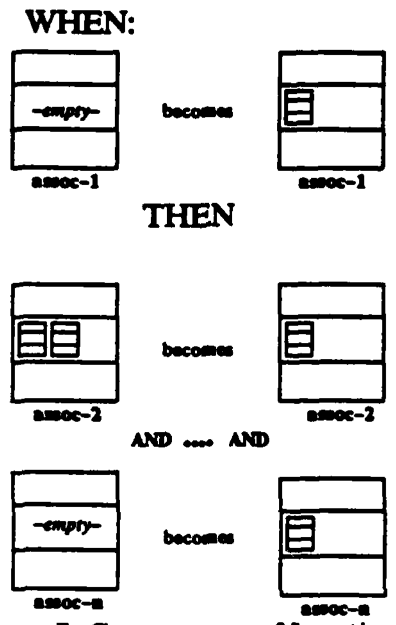


Figure 7: Consequences of Inserting

interpreted as holding true; in this case, ISFI collects a set of constraints that must be made true. In order to collect the constraints that must be enforced, ISFI must examine the role descriptors for *necessary* and *sufficient* conditions. These are predicate nodes in the domain knowledge base that determine an object's membership in the role of another object. Necessary and sufficient condition nodes have the interpretation that when an object is added to the role of an object, all of that role's necessary conditions become true. When an object is removed from the role, the role's sufficient conditions become false. Necessary and sufficient condition nodes can be thought of as specifications of daemon behaviors (when an event occurs, it may cause another event to occur also) but are actually non-procedural (when a role relationship changes, enforce some other relationships). Expressing the necessary and sufficient conditions in a logical form:

$$Role_1(Assoc_1, Member_1) \vee \ldots \vee Role_m(Assoc_m, Member_m) \leftrightarrow$$
$$Role_{m+1}(Assoc_{m+1}, Member_{m+1}) \wedge \ldots \wedge Role_n(Assoc_n, Member_n).$$

To find consequential mutations, an area of network around the necessary and sufficient condition nodes in the domain knowledge base is copied into the problem network and the changing node values are propagated in the network to trigger constraint laws that perform mutations. Any such laws that are found are used in declaring new (consequential) mutations. In this way, the relationships directly determine the additional mutations that are necessary to maintain consistency. Of course, these consequences are also mutations and may have further consequences.

This approach has two noticeable problems. First, a constraint that needs to have its relationship enforced usually has a number of laws that claim to do so. One law may add objects to a role while another may delete them. The two laws can lead us to conflicting sets of necessary and sufficient conditions to maintain. For example, suppose a boolean node is used to trigger a constraint with two laws that conflict as described. If the boolean value is true, the first law is triggered; otherwise the second law is triggered. The actual value of the boolean may not be known until the synthesized program is actually executed. Thus, the synthesized program needs to be prepared for both possibilities. Since either possibility can lead to more consequential mutations, the ISFI system must write a program that for every declared change, walks a tree of consequential mutations with each branch based on the value of some node. In this way, the ISFI system compiles special case truth maintenance code in every program whose specification includes a mutation. Luckily, experience indicates that the trees are typically quite small and in many situations the value of a node is a constant and ISFI can prune the tree at synthesis time.

The other problem with our scheme of mutations is much worse in theory but actually less of a problem in practice. The potential problem arises because ISFI's condition nodes lead to sets of relationships that may have *no procedural interpretation*. Because propagation can fail to find a computation path through a section of network and because we rely on propagation paths to find constraints that need to be enforced, our technique runs the risk of producing an incomplete set of consequences. Even using the transformations, inheritance, and other mechanisms that introduce computation paths, we can never guarantee that constraint consistency has been entirely restored or even that the system has exhausted all reasonable inferences. Our experience indicates, however, that the user is not overly burdened in augmenting the specification so that the system can find important consequences.

3.5 Complex Relationships

We want to group certain sets of relationships into a larger, more complex relationship. In this way, we can build ever more complex relationships and create various levels of abstraction. This capability is achieved using *complex constraints*. Complex constraints are defined using networks of constraints so that a single constraint is used to embody an entire network of knowledge. There are at least five different ways a complex constraint may be used.

The first is to copy the network that defines the complex constraint type into the network where the constraint is being used. This technique is analogous to macro expansion and defines the semantics of the complex constraint. Each of the other four techniques for using complex constraints must produce programs with the same behavior as this technique.

The second technique is to plan the computations in the complex constraint network and then copy over the part of the network that is needed for a particular application. This technique would be useful when the complex constraint network is expressed non-procedurally (is missing a required computation path). A suitable computation path could be found in the complex constraint network once and thus avoid the tedious problem solving that would otherwise occur every time the constraint was used.

The third technique proceeds through finding a computation path and then finding consequential

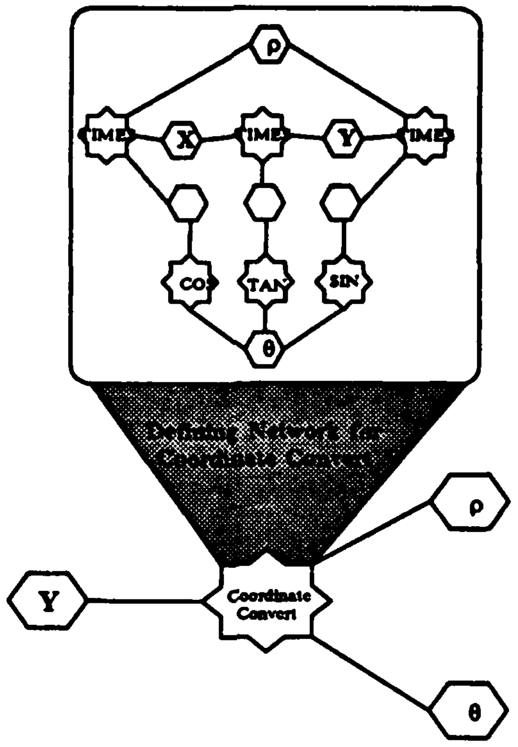


Figure 8: Using a Network to Define a Complex Constraint

mutations. We may choose to find all the consequential mutations that are indicated by the constraint's network. However, we can not guarantee that we will find all of them. For example, a DISPLAY-OBJECT constraint can not possibly know all the details of displaying all objects on all display devices. The system must consider how the constraint is used each time it is used with a set of objects from different classes.

Regardless of whether or not we handled consequences, we can choose to proceed to finding data structures for the nodes in the constraint's network. In many cases, however, we would not want to make data structure decisions until we knew how the constraint is used. For example, the nodes in an application may already have data structure commitments. Any decisions we make about data structures for the complex constraint must be consistent with those commitments. This is a good example of the trade-off between generality and economy of computation.

Finally, we can synthesize an entire sub-routine for each of the constraint's laws. For this technique, the synthesized code that derives from the constraint law is just a function call. Note that even if the constraint's laws are refined to this degree, we may still have some consequential mutations to find or some data structures to include. Also, if the same constraint law is used by an example that demands different data structures, ISFI must re-write the law to fit the new use. This is the only strategy that is currently implemented.

An example of the use of a complex constraint can be found in a number of conversion programs. For example, a program may need to convert a Cartesian coordinate pair (x, y) to the polar coordinate pair (ρ, θ). Rather than specifying the relationships among the four variables everywhere needed, a complex constraint (call it COORDINATE-CONVERT) can be formed with the relationships expressed in the constraints of its defining network. The necessary laws can then be written that compute each of the variables from some combination of the others. Note that only a few relationships are needed because each relationship can contribute in a number of computations. The fact that, "$y = \rho * sin(\theta)$" can be used to compute any one of y, ρ, or θ given the other two.[3] The laws that are written can then be used by propagation whenever a constraint of type COORDINATE-CONVERT is found in a network. Even better, should we decide to express more relationships among the four variables (assuming we had missed one or more that make the computations more eloquent), we could now add them in the one network and know that they could be used everywhere constraints of type COORDINATE-CONVERT appear (see figure 8).

3.6 Overall Control

Overall control of the automatic programming system is handled by the *Agendas* [5] problem-solving and planning mechanism (see figure 9). This mechanism has available a number of standard plans for problem solving strategies, meta-plans for choosing the correct plans, an ability to make assumptions, and replanning facilities for recovering from failures. Agendas were designed specifically for controlling the ISFI system and later extended [21] and used in two applications [22,23]. Agendas work under the assumption that all possible operators in a domain are known. For the ISFI domain, the operators are the mechanisms such as propagation and transformation. The Agendas controller attempts to satisfy a set of goals by selecting among the available operators. Agendas are not necessarily tied to any given strategy (such as "best first," GPS, "hill climbing," etc.) for applying operators but uses metaplans to represent various strategies. Another advantage of Agendas is that they support clear, declarative definitions of strategies so that each strategy has easily recognized consequences.

As applied to the ISFI system, agendas attempt to satisfy four major goals in order: determine a rough algorithm, construct data structures, compose a concrete algorithm, and generate code. Determining and composing algorithms can both be accomplished by propagation. If propagation fails, the agendas select other mechanisms that may enable propagation to succeed. Also, goals can have subgoals (e.g., expand or satisfy complex constraints before composing a concrete algorithm). Finally, some goals may interfere with others. For example, the set of known data structures may not support the operations required by the initial, rough algorithm. Thus, the second goal may cause the agendas mechanism to backtrack and attempt to find a different algorithm.

[3]This is actually an oversimplification because θ is not unique for some values of y and r.

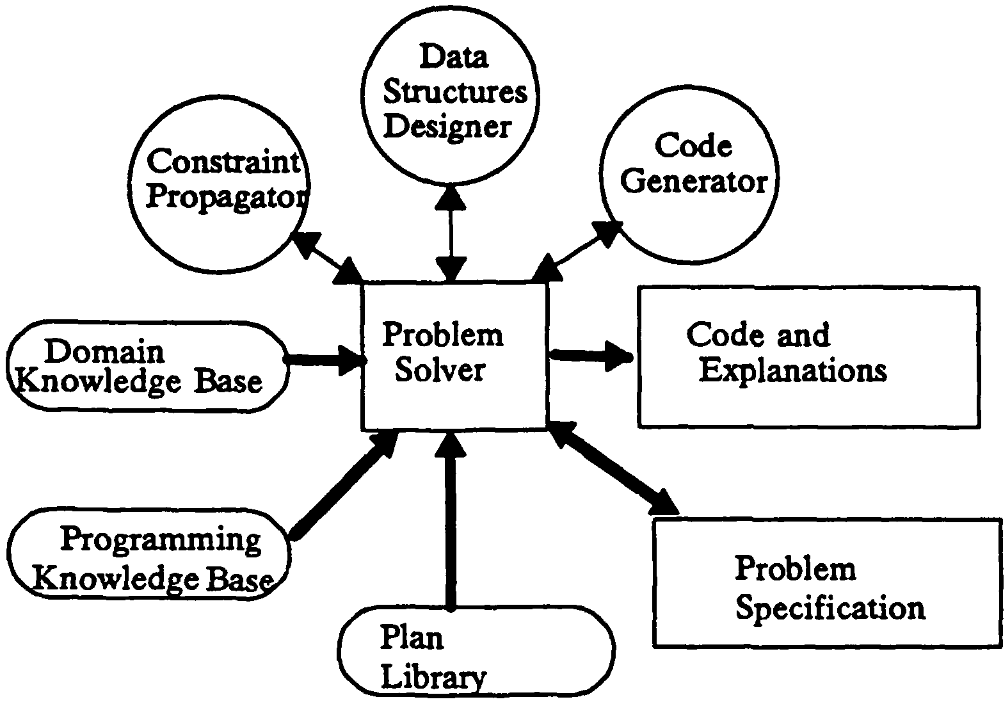


Figure 9: ISFI System Architecture

3.7 Deriving Data Structures

ISFI needs to map the abstract objects to structures inside the computer. Real world objects may be represented in the synthesized program as variant records, as linked lists, as vectors, or as other data structures. Every real world object must be represented as some programming language (implementation) object in order to participate in the operations derived from the constraints.

Constructing these representations generally consists of collecting some number of roles for an object class and mapping those roles to various parts of a data structure. The process of finding suitable mappings can get very complex, especially when considering ordering and indexing or when some roles are either *clustered* or *spread*. Clustering combines different roles into the same field of a data structure (e.g., the array size stored as the first element of the array). Spreading is when a given role occupies various fields of the data structure (e.g., representing a role's index in a separate data structure from the role's data). Also, some roles that could potentially be included in a given mapping may not be. Determining which roles to use (and how) is a general design problem of deciding what attributes of a given object are relevant to a particular program. For example, based on the computations in a program to track drug smuggling, one could decide that the color of a fishing boat is not important (not referenced by any computation) and may be omitted.

Consider objects of class COLOR where we want to express three components (roles) of every color - the red part, the blue part, and the green part. Obviously most programming languages do not have data types called colors. It is possible, however, to build or use a data type already existing and call it

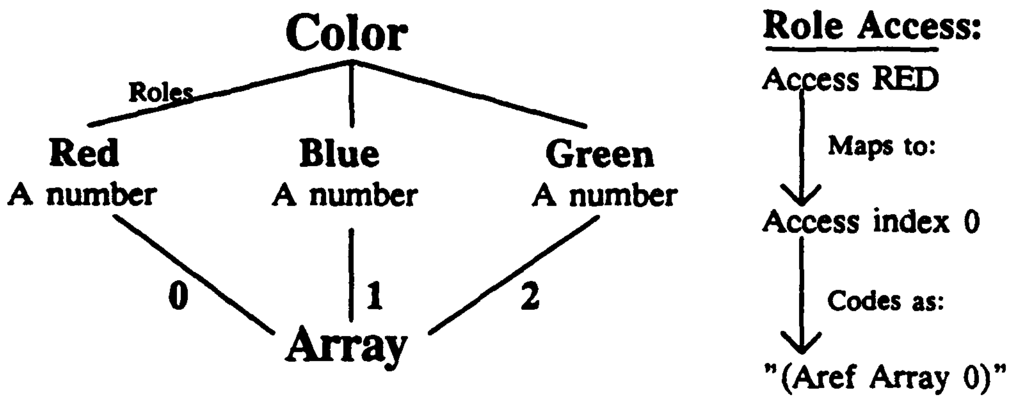


Figure 10: Mapping Operations to Code

a color. One might choose, for example, to build colors out of arrays of length three where each element holds one of the parts. Representations record such a decision.

With the mappings in place, we can map an operation on colors to an equivalent operation on arrays. For example, accessing the red part of a color can be transformed to accessing the first element of the array (see figure 10). Code is then produced that implements the operation in the programming language for that data type. In this way, objects acquire the capabilities of the data types they are built on.

ISFI also allows an object to be represented in terms of other objects. Here, the mappings are set up from the first object and its roles to the second object and its roles. Operations to be performed on the first object are mapped to operations on the second object which must then map the operations to somewhere else (either yet another object or a data type). In this way, objects can also gain the capabilities of other objects.

As an example, we may model a star as a point on the celestial sphere and represent the star in ISFI as a point in two dimensions. We may go on to represent the star's apparent motion as a vector. ISFI knows that points can be represented as vectors so the stage is set to compute a new position of the star merely by adding vectors – an operation ISFI knows how to implement.

4 Code Generation

4.1 Building Blocks

The basic unit for code synthesis in ISFI is a *building block*. Building blocks contain the information that makes the generation of code possible. Building blocks are the way-points between the state transition hierarchy and the actual code. They not only provide a way to organize information, but are the embodiment of ISFI's language independent capabilities.

Building block types represent the abstract control structures present in all programming languages (conditionals, subproblems, iteration, sequence, etc.). For example, an iteration is the same whether implemented using GOTO's, tail recursion, or highly structured WHILE-DO constructs. A tree of building blocks, called the *program structure tree* is an abstract representation of control flow and block structure that satisfies the specification. Because ours is a language independent model, the building blocks and the program structure tree express the meaning of the specification in terms of the lowest common denominator of the syntax and semantics of various programming languages. The building blocks also act as back-pointers from the code strings to the specification and provide the explanation facility with much of the needed information to generate useful explanations.

4.2 The Building Block Hierarchy

Building blocks are implemented using the object-oriented aspects of Symbolics Zetalisp [24]. Different control construct types are organized in a hierarchy. The basic types of building blocks are:

sequence block: An abstraction for a sequence of statements and/or expressions that are ordered according to data flow.

case block: An abstraction for conditionals; generalizing behavior of if-then-else statements and case statements.

loop block: An abstraction of all iterative behavior.

program: An abstraction for a procedure.

break conditionals: Used to abstract the halt conditions of all iterations. Break conditionals must be associated with loop blocks.

calling blocks: An abstraction for procedure calls.

For each language for which code is generated, there are language specific types of each of the building block types (i.e., lisp-loop-block, c-program, etc.). In addition, various behaviors can be mixed into the

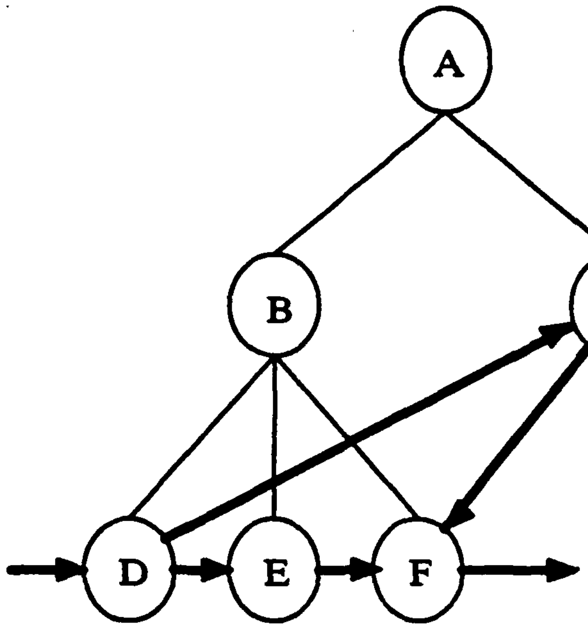


Figure 11: Need to Re-establish Environments

building blocks of particular languages by using the facilities of multiple inheritance available in object-oriented programming. For example, different languages allow different constructs to scope (declare) variables. (Loops can scope variables in Lisp, but they cannot in C). In this way, for a given target language, ISFI is able to select only appropriate behaviors and organize them conveniently.

The information described in each building block includes: the variables to scope and assign, the side effects that take place, and data flow information concerning computations local to the building block's environment. This includes the inputs and outputs and a data flow graph.

4.3 Creating the Program Structure Tree

Our theory of code generation depends greatly upon the generation of the program structure tree (PST). The program structure tree is essentially a distillation of the meaning of a specification. Subproblems are discovered, commitments are made for control flow, and scoping environments are created. In this way, what is implicit in the state hierarchy is made explicit.

Some commitments are obvious: decision points become case blocks, subproblems can occur when two paths in the state transition graph meet, etc. However, because of the freedom provided in our model, the creation of scoping environments and the enforcement of consistency (for side effects) are not trivial.

In the specification in Figure 11, the state C is not in the same environment (branch) as the state F. The transition from C to F could cause an inconsistency if the correct environment is not re-established.

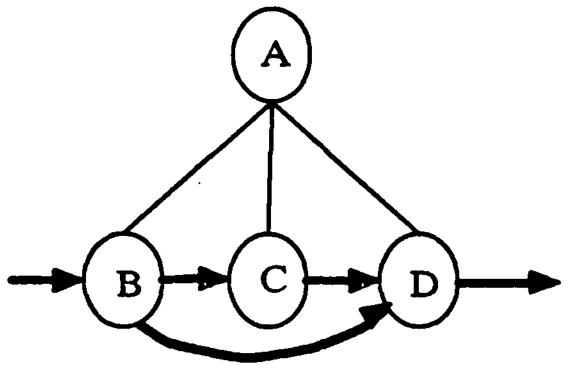


Figure 12: Environments of the Decision Point and Clause are Unrelated

In the specification in Figure 12, the clauses of a conditional are unrelated to the environment of the decision point. This is a reasonable way to write a specification, but is contradictory to the semantics of conditionals in most programming languages.

To create a program from the specification ISFI must make commitments to the exact control flow and scoping environments. This means it must find an interpretation of the state transition hierarchy that is complete and consistent. Once the behavior implicit in the state transition hierarchy has been determined, ISFI can alter the scoping implied by the specification by introducing new control flow information.

How a program makes use of a state is expressed by *state usages*. A state usage captures the environment of part of the network specification. Finally, it is these state usages that are associated with building blocks. State usages are composed primarily of *state visitations*. Three aspects of a state are organized by a state visitation: the incoming and outgoing transitions and the state's location in the hierarchy. A state visitation summarizes a way to "visit" the state. In other words, a visitation expresses a single path through a state.

The first example above illustrates a situation where it is necessary to *split* a state. Splitting a state into two or more state usages is necessary when a state has state visitations that are not *homogeneous*, thus causing some inconsistency in the use of a state's environment. Homogeneous visitations are the normal case and do not cause any inconsistencies. There are three reasons for an inconsistency:

- Entering a state from different branches of the state hierarchy.

- Using wires to import and/or export different objects' values.
- Non-equal sets of side effects.

Note that wires and side effects are associated with transitions into a state. The set of visitations is partitioned according to these three factors so that each state usage is self-consistent.

From the state usages, ISFI derives the various control structures in the diagram. The inconsistencies illustrated in the second example are handled at this time.

First, cycles in the graph identify the positions of loops. ISFI supports a notion of structured programming so only cycles that have one entry point are considered loops. (Other cycles are treated as recursive subproblems.) The state usage forming the entry point of a cycle is associated with a loop block building block. All state usages that are exit points from the cycle are associated with break conditional building blocks. The use of break-conditional blocks allows the introduction of multiple exit points in loops.

Next, all decision points not involved in loops are associated with case block building blocks.

All remaining state usages with more than one incoming transition are associated with program blocks, unless they are *join points*. A join point is the state usage where the paths from a conditional meet or the destination of all transitions out of the loop. Calling blocks represent the callers of program blocks and are not associated with any state usage. This not only allows us to organize the procedure calling behavior, but also permits the graph from the state transition hierarchy to be transformed into a program structure *tree*. Traversing trees instead of graphs reduces the complexity of many algorithms in code generation.

Finally, any remaining state usages are associated with sequence building blocks. A sequence building block contains various operations whose time ordering is based solely on data flow information.

4.4 Variables

ISFI manages the use of several types of variables. As with building blocks, variables are organized into a hierarchy of types using object-oriented techniques. In keeping with the spirit of language independence, there are language specific types for each variable type (e.g., c-output-variable, lisp-transmission-variable, etc.). This aids in the generation of code because various languages implement variable types differently. For example, output variables (variables introduced to return values from procedure calls) need not exist in Lisp, but they make the code of a C program more readable. Further, various languages support different value passing abilities that require a fairly sophisticated, language dependent capability (e.g., Lisp allows values to be passed directly out of IF statements whereas Ada does not and needs to use variables to make up the difference).

The variables used to pass computations between the various contexts found in the specification are called ***transmission variables***. ***Local variables*** are used to avoid redundant computations.

4.4.1 Sharing Information Without Using the Hierarchy

The state hierarchy allows a specification to access objects lexically. All objects in ancestor states are accessible, and can contribute to computations in descendent states. Objects in sibling states can not be connected directly via constraints because they are in different contexts. The existence of a wire implies that the computation must be saved before switching contexts. This activity appears in the generated code as an assignment statement for a transmission variable. The variable is scoped in the generated code in the environment that includes all contexts using the computation.

The sharing of computations becomes more complex when considering a state usage that corresponds to a program block. Not only must we consider the computations associated with the wires (corresponding nicely to formal and actual arguments of the subroutine), but the calling environment (which appears in the program structure tree as a calling block) could differ radically from the subproblem's own environment. Variables must be introduced to maintain the environment implicit in the specification.

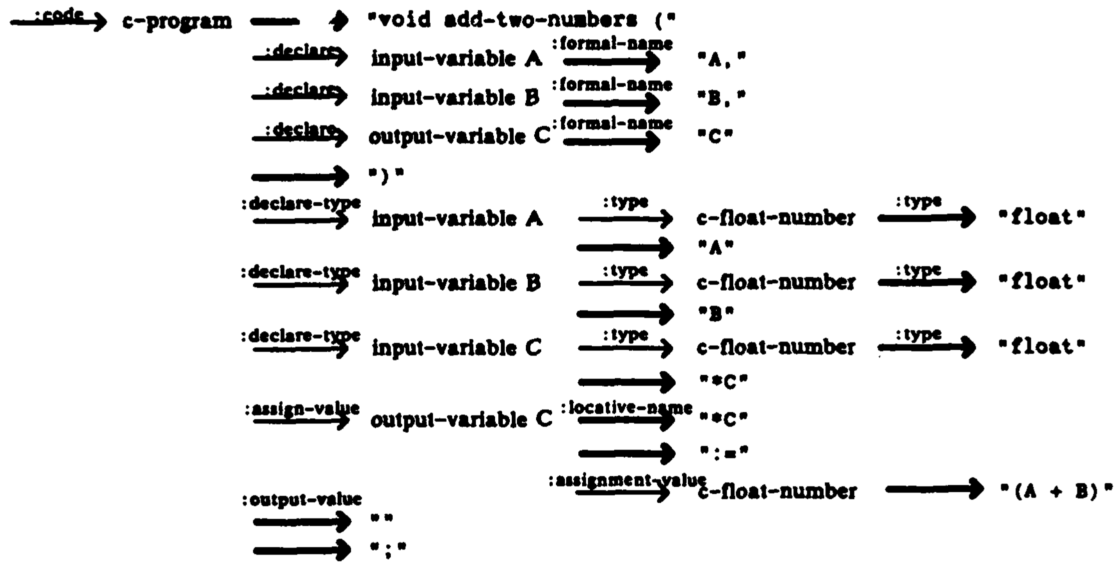
4.4.2 Avoiding Redundant Computations

Each computation records its dependencies and the computations that depend on it. In this way, one may easily construct a data flow graph. Finding redundant computations is one of the purposes of this graph.

There are four ways that two computations can be equivalent:

- They are equal.
- Computations may be copied arbitrarily.
- Part-for-whole and whole-for-part degeneracies. Because all objects in ISFI are aggregates, some computations are introduced when accessing the sole part of an object.
- The computations could be semantically equivalent (e.g., $(y \times (2+x))$ is the same as $(y \times 2)+(y \times x)$). Obviously, to spot such similarities in general is undecidable. ISFI identifies certain broad classes of these similarities.

Except for those prohibited by side effects, all computations that are equivalent (or transitively equivalent) in these ways are made to share variables.



```
"void add-two-numbers (A, B, C)
    float A;
    float B;
    float *C;
  *C := (A + B);
  end;"
```

Figure 13: Code Generation in C

4.5 Producing Code Strings

ISFI's implementation uses program building blocks, implementation objects, and variables to produce code strings. These three kinds of objects are called *coding-objects*. Each coding-object produces code strings by responding to certain messages. [4] We invested considerable effort to make this a straightforward affair even in the face of multiple target languages.

As stated above, every coding-object has a set of messages that it handles by producing code fragments. For example, a variant record implementation object has the *access-a-field-by-name* message that produces code to reference a field of a particular object implemented as a variant record. The particular object and the name of the field (usually derived from the name of some role of the original object) are passed in as arguments to this message. Similarly, variables can produce code to assign themselves to a given value and case blocks can produce the code for each of their clauses by sending more messages. This process is represented pictorially in figures 13 and 14. Messages names are prefixed with a colon and generated code strings are in quotes. All other text represent coding objects.

In other words, ISFI derives object representations and operations to be performed and finally orders and digests the rough sequences of operations in building blocks. At this point all the hard work is done; one obtains code by sending a message to the top-level building block (which sends messages to its variables, inferiors, etc.)

[4] We use the Symbolics Zetalisp object-oriented programming terminology in this section. Those unfamiliar with this vocabulary can think of messages as generic functions, and sending messages as a generic function calls.

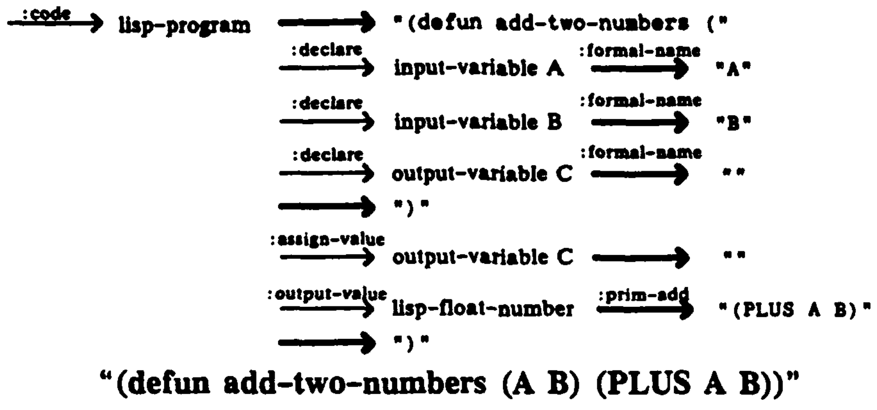


Figure 14: Code Generation in Lisp

In reality, code fragments are more complex than simple character strings. Recalling that each code string is created by sending a message to a coding-object with certain arguments, ISFI can also annotate the code with the object, message, and arguments. This annotation allows the system to trace the production of code back to the coding-objects and operations. ISFI only reasons about the annotation, never the character string itself. From coding-objects and operations, the system can then trace back through the various mappings to specification-level objects and constraints to explain the code in terms of the user's original statements. For example, some code and *its annotation:*

Code:``(plus A B)''

Implementation Object: Number-Implementation

Abstract Object: Color-Peak-Frequency

Message: :primitive-add

Arguments: numbers A, B

Explanation: Adding numbers A and B because the specification references the sum of the spectral distributions of two colors.

5 Interface

The primary means of entering specifications for the ISFI system consists of using the text editor to manually create and connect network constructs such as nodes, constraints, and wires. Not surprisingly, the interface needs to do more than just accept specifications. One must be able to synthesize code, examine the work of the various mechanisms, and review the network specification after it has been entered. To better serve these purposes, a graphical interface was developed [25]. Note that this interface was constructed for use by the system developers not by the eventual users (the system has never had any "pure" users). Thus, a much more complete ability to access the system must be provided than merely editing specifications and receiving synthesized code.

In the Gist English Generator [26], an English paraphraser was introduced to aid in the understanding and debugging of specifications. ISFI's graphical interface fulfills much the same purpose. By drawing networks of constraints in the same spirit as the figures shown above (e.g., Figure 1), users benefit in that having two means of examining a specification lends extra confidence that the specification will behave as intended. It is our experience that one more easily assimilates the meaning of a set of relationships by viewing a graph presentation than by viewing text. The use of the connectivity of the graph itself is extremely helpful. It provides a quick and easy way to navigate through the specification. Questions, such as, "how is this object changed," "what other objects is it related to," are easily answered by traversing the graph. The graphical presentation also allows one to view only that part of the specification that one feels is relevant at the moment. Further, the position of the nodes and constraints can be manipulated so that related objects are closer while unrelated objects are farther apart. The graphical presentation helps to reinforce the focus in a way that static text can not.

The ISFI interface is not limited to viewing networks of constraints. The interface also provides capabilities for editing specifications, writing programs, invoking mechanisms manually, changing network structure, delving into the knowledge bases, and viewing the synthesized code and code explanations. Most of this is accomplished using a menu-driven command processor. Additional windows become visible for special purposes such as viewing the several kinds of hierarchies ISFI maintains or for examining the "before and after" nature of transformations. The KingKong [27] natural language interface has been ported to the ISFI system and is used for question answering about the specifications and synthesized code.

6 Real-Time Automatic Programming

The technical objectives for the FY87 automatic programming project included plans for addressing the issues that arise when one attempts to produce real-time programs. However, after the first quarter of the year it became apparent that despite the amount of research into so-called real-time programs and multi-processing systems, there exists little agreement among researchers as to what constitutes a real-time program, how to formalize the issues of multi-processing (or asynchronous behavior in general), and, most importantly, how to synthesize programs that effectively utilize multi-processing environments. Research into real-time programs was not as mature as we had believed.

Only two groups in the country have made any headway into synthesizing real-time programs. Barstow [28] has presented a method of implementing programs based on the theory of stream processing. The presented results of this work are limited in the application domain and extremely specialized in the type of programs produced. Green, King, et al [29] regularly report on their theories of concurrent program synthesis. While designed to be widely applicable, their results have only been tested in a limited setting. Both groups fall short on the level of specification spectrum, although Green, King, et al, are working seriously on raising that level.

Our primary result in using the ISFI system to address real-time (multi-processing) software is that we have begun to define the areas of interest and challenge. The remainder of this section details what we feel would be necessary in order to make ISFI synthesize real-time programs. For the most part, real-time programs are addressed in terms of their execution time. The same statements could be made concerning memory requirements.

6.1 Definitions

The execution time of a program segment is the time measured by a standard time clock (not, for example, by CPU instruction cycles) between the beginning and end of the execution of the program segment. While this sounds trivially straightforward, note that this is not the same as the time the program segment spent actually executing. Much of the execution time of programs written on current hardware is consumed in waiting for other activities to finish. The execution time of a program segment must include activities such as waiting for a virtual memory system to page in a necessary block of memory, waiting to resume after interruption by other programs, waiting for the completion of an I/O operation to a user, tape drive, etc.

A real-time program segment is a segment of executable code that has a guaranteed maximum execution time. Typically, only some segments of a program are so guaranteed because many program actions, such as waiting for a user to respond to a query, have no upper bound on their running time. For this reason, the truly time-critical sections of software are usually protected from interrupts, user input, and other unpredictable disturbances.

A process includes the program and the data required to execute the program. Some data may reside in several processes. A process is "active" if it is ready to run, i.e., has all necessary data and programs available. A process is "running" if the process' program is currently utilizing the CPU. The execution interval of a process is that period of time during which the process is running.

A multi-processing environment is a computer configuration (hardware and software) that allows more than one process to be active at the same time. Parallel processor environments are a distinguished subset of multi-processing environments because several processes may be running at the same time.

6.2 Specifying programs in ISFI

A real-time program specification in the ISFI system is nearly identical to the specifications for the types of programs that ISFI currently addresses. The only change is that the user must have some way of specifying the maximum execution time (or memory utilization) of the program. Presumably this addition could be outside the usual context of objects and relationships that programs address.

By contrast, the specification of a multi-processing program (or set of programs) takes on a whole new aspect. This is because ISFI supports the specification of program behavior where program behavior is limited to manipulating the objects that the program can reference. In specifying multi-processing behavior, we must now consider the processes as objects and broaden our vocabulary to allow programs to manipulate processes. Specifically, ISFI must support specifications of the relationships among processes as they change over time. With the level of complexity and detail that this entails, ISFI must provide a more sophisticated means of specification than the "just one more consideration" approach mentioned in the previous paragraph.

Relationships among processes involve passing data, sharing data, relative execution intervals, and parent/offspring relations. For example, a process may provide data to another process either through parameter or message passing. Similarly, a process may preempt another process, preventing it from running, or create a new process and set it on its way. Since the relationships among processes vary over time, ISFI's context mechanism (states) must also be adapted to this new discussion.

Since ISFI represents relationships in networks of constraints, one envisions creating separate networks for specifying the relationships among the objects that the program manipulates and for specifying the relationships among the processes. However, the two specifications are not independent with respect to their state diagrams. For example, a process may become inactive (represented by a transition in the processes' state diagram) after the final state of the program's state diagram has been reached. In other words, the two state diagrams (at least) need to communicate.

We considered a number of ways that this communication might be achieved. We explored the idea that transitions among the states could carry additional information such as enabling interrupts, waiting for events, occurring periodically within a certain time, etc. We explored partitioning the state hierarchy into various sections that could have attributes (e.g., a set of states might represent a process). Keeping in mind that whatever communication scheme we arrive at must support code synthesis, no scheme presented itself that was relatively simple and had a clear semantics of program specification.

In general, what is needed is a way of grouping together networks of constraints and expressing relationships between the networks. The progress made on this issue through the year was primarily to define the issue. Future work should explore more concrete methods for addressing these communication needs.

6.3 Necessities for Synthesizing Code

ISFI achieves its success in code synthesis by relying on the common properties of a large group of programming languages. In this way, the same program can be written in various programming languages by first mapping into an internal representation of the program and then producing code from the internal representation. For example, the relationship stating that certain words were present on a monitor's screen could be enforced by producing an operation that would place the words on the screen. This operation is represented precisely as stated ("Place text on monitor's screen") in the internal representation. For a particular programming language and monitor this operation may be translated as, "Print text on monitor," (if the language directly supports the operation we need) or as, "send 'draw-text' message to monitor," or even as, "store ASCII characters at memory location 32516."

We had envisioned much the same arrangement for carrying out the operations of a multi-processing environment. Example operations are, "Disable interrupts," "wait for event," and "provide data to other process." The problem with adopting this approach is that, unlike the operations mentioned above, there are very few generalized operations to apply in the multi-processing environments. That is, the internal representation of the operations would not be far removed from those actually understood by the particular environments. Of the set of operations that are most commonly discussed in the multi-processing domain, most multi-processing environments support only a subset.

By way of analogy, suppose that our target environment does not directly support the "place text on monitor's screen" operation. With so little generality to draw upon, chances are that our target environment will also not support the other two operations either. This burdens us with the much harder problem of finding conversions between operations that are not very much alike. For example, we may need to use polling techniques to implement the "wait for event" operation. However, polling will affect the scheduling of all involved processes. Some other process may need to "disable interrupts" for a time and this will not automatically work in a polling strategy. The ISFI system would somehow have to provide not only the polling scheduler but the modification that allowed one process to monopolize the processor for a given time. In other words, to be most general, ISFI would have to augment the operating system itself.

The point is that we have suddenly stepped into a much messier situation where we can not make use of strategies that have worked in the past. We discovered this unsettling situation only upon attempting to apply our proven methods. Instead of translating general operations to those supplied by a given programming language, we now face the task of translating specific capabilities intended to achieve certain goals into equally specific operations created with other goals in mind. Certainly, any given strategy for a given environment can be produced but the combinations of strategies and environments is overwhelming.

6.4 Rebuttal

Through the year, we considered several alternate strategies for directing the research being done. The main concepts are presented below with the reasons why we rejected them.

- Reduce the insistence on producing code for various target languages and operating systems. Provide more power over a narrower scope.

  The Air Force, MITRE, industry, and others all produce software in a variety of environments. Limiting ourselves to a single language, running in a single system, would severely decrease the value of our work. This is especially true when one considers the possible target languages and environments. Using the language C in Unix is the closest thing to a universal environment and, as such, would not be so terrible. DOD, though, has mandated the use of Ada without reference to operating system. This mandate makes one hesitant to pursue the Unix environment whereas Ada's tasking capabilities are extremely lacking in the power to carry out complex tasking behaviors. Neither possibility begins to address the number of special hardware configurations that the Air Force regularly pursues where no operating system to speak of is available (e.g., JTIDS secure voice digital radio).

- Lower the level of specification.

  The automatic programming community recognizes a spectrum of specification levels ranging from machine code through high-level programming languages (Lisp, C, Ada) and onward to abstract specifications of behavior. It is at this high end that the ISFI system accepts specification, taking on the burden of translating to less natural, less general, more concrete specifications known as code. Although nearly every automatic programming effort strives toward the high end of the spectrum, most begin lower for the sake of near-term results.

  We chose not to pursue this strategy for two reasons. First, it would reduce the ISFI automatic programming system to the level of a programmer's aid or even to the level of "just another programming language" and, again, substantially limit the value of our research work. Secondly, even this strategy for achieving a near-term payoff at the expense of the more theoretical work could not be completed in less than several years regardless of how far we were willing to lower the level of abstraction. In other words, this strategy would only be appropriate for a moderate range program that was interested in building a robust working system for production use.

- Give up or reduce the requirement of an internal representation of operations to be performed.

  Basically, this strategy would require that the common internal representation be made more abstract rather than abandon it. Behaviors would be the common ground rather than operations. For example, the notion of "execute until interrupted" could be implemented in the context of most of the more concrete strategies that are directly implementable (polling strategies, prioritized interruption schemes, even Ada's tasking capabilities).

  In the long view, and assuming no one develops a general theory of multi-processing that is compatible with our needs, this is definitely the solution to seek. There is no insurmountable obstacle to progress in this direction. It does, though, require a major, high level revision of the ISFI system's method of program synthesis. This belief is contrary to those held when proposing this project.

Though this last option progresses along a desirable course, it amounts to producing the general theory that is currently lacking in the computer community. Such a result, already widely sought, could only come from of an intensive, multi-year effort with clearly defined goals. Only near the successful conclusion of such work would it be appropriate to begin the true automatic programming work. Certainly, many unforeseen issues will continue to appear.

7 Related Research

There exists a large body of automatic programming literature which the ISFI project has drawn upon. Both Steele [12] and Borning [13] investigate the uses of constraint languages for knowledge representation. In both cases, the intent is to "interpret" the networks rather than to "compile" them as ISFI does. ISFI also treats the notion of context differently from either.

We share the view presented by Balzer [4] that the most important phase of software production is the maintenance phase. However, we do not support symbolic execution of specifications as presented by Cohen [30]. In [30], symbolic execution was presented as a way to perform rapid prototyping without fully synthesizing the specified program. Since we believe it possible to automate most of the programming details, we prefer to address rapid prototyping by synthesizing complete programs from partial specifications. In other words, we pursue the same goal of letting the user examine his incomplete specification "in action" but with the system more able to provide complete programs.

Our work in recovering from propagation failures by bringing more knowledge to bear resembles both the work by Green [31] and Barstow [32]. Barstow argues that a wealth of domain dependent knowledge must be used to synthesize good programs. We agree but further believe that a general set of mechanisms can make use of specialized knowledge from a number of domains. The work described by Pressburger [33] begins with an advanced programming language and continues to build more "non-procedural" concepts into the language (e.g., logic specifications, the MAINTAIN statement). Much of its non-procedurality is resolved by bounding set memberships and has a different flavor from the non-procedural specifications found in ISFI.

ISFI differs from other automatic programming systems in using separate high and low level representations of programs. This distinction facilitates target language independence. The more common approach uses a single "wide-spectrum" representation of the program being produced. Most researchers have emphasized the problems of algorithm synthesis and data structure selection over the issues of code generation (e.g., [16]). The ISFI project is equally concerned with both algorithm and data structure issues. ISFI's program structure tree resembles the Program Apprentice's plans [34] in attempting to describe common programming constructs rather than the "first principles" (e.g., goto, return, address memory location) description of many systems. Barstow [32] briefly mentions the need to create actual source code from the algorithmic description but implies that because the representation of the program follows so closely the conventions of the programming language, the translation is trivially straightforward. Similarly, Kant [35] develops the program down to the fine details with no break between committing to an algorithm and improving it in "small" ways. In ISFI, due to the requirements of language independence, the process is not as simple. However, the work in [36] is sufficiently general that it addresses many of the issues the ISFI system has faced. Comparisons of our code generator can be made to compiler technology [37]. These are not entirely appropriate since compilers seek to create machine-specific program structure from text while code synthesis seeks to generate text from abstract program structure.

8 Conclusion

The problems implicit in describing multi-processing behavior in such a way that one could automatically synthesize real-time program code are much harder and less well understood than we realized. Future programs should have modest, well defined goals and expect to realize those goals only over relatively long periods.

In lieu of concrete results in the real-time domain, we spent considerable time this year working on making ISFI a robust system. Producing code uniformly in a variety of programming languages is one of the system's strongest results. This was enabled by the design and implementation of a new code generator for ISFI. We feel that the system is now in an excellent state to continue research in the future.

References

[1] R. Brown, *Coherent Behavior from InCoherent Knowledge Sources in the Automatic Synthesis of Numerical Programs*. PhD thesis, Massachusetts Institute of Technology, January 1981.

[2] G. Sussman and G. Steele, "Constraints - a language for expressing almost-hierarchical descriptions," *Artificial Intelligence*, vol. 14, 1980.

[3] D. Barstow, "Artificial intelligence and software engineering," in *Proceedings of the 9th International Conference on Software Engineering*, March 1987.

[4] R. Balzer, T. Cheatham, and C. Green, "Software technology in the 1990's: using a new paradigm," *IEEE Computer*, vol. 16, November 1983.

[5] R. Brown, "Automation of programming; the ISFI experiments," in *Expert Systems in Government Symposium*, (K. N. Karna, ed.), IEEE, October 1985.

[6] M. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision*, (P. Winston, ed.), McGraw-Hill, 1975.

[7] D. Touretzky, *The Mathematics of Inheritance Systems*. Los Altos, CA: Morgan Kaufmann, 1986.

[8] R. Brachman and J. Schmolze, "An overview of the KL-ONE knowledge representation system," *Cognitive Science*, vol. 9, 1985.

[9] R. Brachman, R. Fikes, and H. Levesque, "Krypton: a functional approach to knowledge representation," *IEEE Computer*, vol. 16, Oct. 1983.

[10] W. Swartout, "Xplain: a system for creating and explaining expert consulting programs," *Artificial Intelligence*, vol. 21, pp. 285–325, 1983.

[11] W. Martin, "Descriptions and the specialization of concepts," in *Artificial Intelligence: An MIT Prespective*, MIT Press, 1979.

[12] G. Steele, *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD thesis, Massachusetts Institute of Technology, 1980.

[13] A. Borning, "The programming language aspects of thinglab, a constraint-oriented simulation laboratory," *ACM Trans. Programming Languages and Systems*, vol. 3, Oct. 1981.

[14] R. Brown, "Agendas: a metaplanning mechanism," M-series M85-26, MITRE Corporation, Burlington Road, Bedford, MA 01730, July 1985.

[15] R. Balzer, "A 15 year perspective on automatic programming," *IEEE Transactions on Software Engineering*, vol. SE-11, 1985.

[16] D. Barstow, "Automatic construction of algorithms and data structures using a knowledge base of programming rules," *SAIM*, vol. 308, 1977.

[17] G. A. Cleveland and R. Brown, "Mutations and their consequences; a study of non-monotonic behavior," in *Expert Systems in Government Symposium*, (K. N. Karna, ed.), IEEE, October 1985.

[18] J. McCarthy and P. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence 4*, (Meltzer and Michie, eds.), New York, NY: American Elsevier, 1969.

[19] J. Doyle, "A glimpse of truth maintenance," in *Artificial Intelligence: An MIT Prespective*, (Winston and Brown, eds.), MIT Press, 1979.

[20] D. McDermott, "Contexts and data dependencies: a synthesis," *IEEE Pattern Analysis and Machine Intelligence*, vol. 5, May 1983.

[21] M. Zweben, "CAMPS: A dynamic re-planning system,". submitted to IEEE Expert.

[22] R. Brown, "A solution to the mission planning problem," in *Second Annual Aerospace Applications of Artificial Intelligence Proceedings*, Oct. 1986.

[23] G. B. Hankins, J. W. Jordan, J. L. Katz, A. M. Mulvehill, J. N. Dumoulin, and J. Ragusa, "EMPRESS: An expert mission planning and RE-planning scheduling system," in *Expert Systems in Government Symposium*, Oct. 1985.

[24] D. Moon, R. Stallman, and D. Weinreb, "Lisp machine manual," Tech. Rep., Massachusetts Insitute of Technology, June 1984.

[25] A. L. Schafer, "Graphical interactions with an automatic programming system," in *1986 IEEE International Conference on Systems, Man, and Cybernetics*, October 1986. Submitted to IEEE Transactions on System, Man and Cybernetics.

[26] W. Swartout, "The gist english generator," in *AAAI-82*, 1982.

[27] C. Kalish and M. Cox, "Porting an extensible natural language interface: a case history," in *Proc. of the Sixth National Conference on Artificial Intelligence*, (Seattle, WA), July 1987.

[28] D. Barstow, "Automatic programming for streams," in *Proceedings of the Ninth International Joint Conference in Artificial Intelligence*, Aug. 1985.

[29] R. M. King, "Knowledge-based transformational synthesis of efficient structures for concurrent computation," Tech. Rep. KES.U.85.5, Kestrel Institute, May 1985.

[30] D. Cohen, "Symbolic execution of the gist specification language," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, IJCAI, 1983.

[31] C. Green, "Knowledge-based programming self applied," *Machine Intelligence*, vol. 10, 1982.

[32] D. Barstow, "A perspective on automatic programming," *AI Magazine*, Spring 1984.

[33] T. Pressburger, "An environment supporting the automation of software development," Tech. Rep. KES.U.86.7, Kestrel Institute, Sep. 1986.

[34] C. Rich, "A formal representation for plans in the programmer's apprentice," in *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, (Vancouver, Canada), pp. 1044–1052, August 1981.

[35] E. Kant and A. Newell, "An automatic algorithm designer: an initial implementation," in *Proceedings of AAAI-83*, August 1983.

[36] J. V. Coursey, "Knowledge-based compilation using finite differencing techniques," Tech. Rep. KES.U.85.6, Kestrel Institute, 1985.

[37] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.